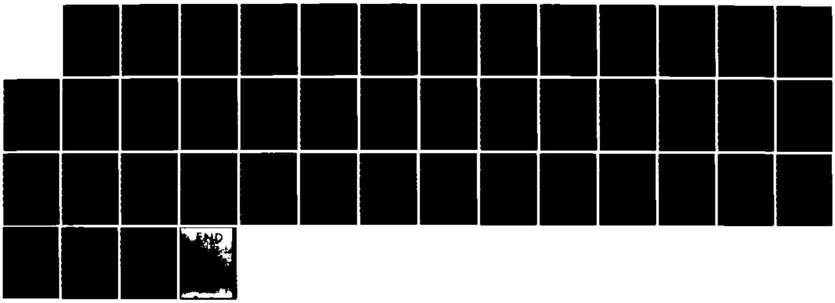
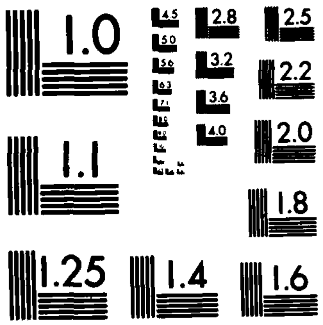
AD-A144 335 A COMPUTATIONAL LOGIC WITH QUANTIFIERS(U) TEXAS UNIV AT AUSTIN INST FOR COMPUTING SCIENCE AND COMPUTER APPLICATIONS R S BOYER ET AL. JUL 84 N00014-81-K-0634 1/1

UNCLASSIFIED F/G 12/1 NL




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

A COMPUTATIONAL LOGIC WITH QUANTIFIERS

Robert S. Boyer and J Strother Moore

DRAFT JULY 1984 DRAFT DRAFT DRAFT DRAFT

The research reported here was supported by National Science Foundation Grant MCS-8202943 and Office of Naval Research Contract N00014-81-K-0634.

Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, Texas 78712

Table of Contents

1. The Formal Syntax

The theory with which the theorem prover deals is presented in this and the next two Sections. This account supercedes and makes obsolete all previous accounts.

A logical theory consists of a language, some axioms or axiom schemas, and some rules of inference. However, in developing the proofs of interesting theorems it is often necessary to introduce axioms defining new concepts and operations. Logically speaking, the main results and all of the lemmas along the way are proved in the final theory. But practically speaking, the theory in which one is working "evolves as time goes by."

To accomodate the practical view of the situation we provide several "extension principles" by which the user of the theory can add new axioms of a particularly constructive sort. Among these principles is the "shell principle," which permits the axiomatization of a "new" type of inductively constructed object, and the "definitional principle," which permits the introduction of an equation defining a recursive function. These extension principles can be considered as rules of inference since they permit one to deduce that certain formulas are theorems.

Our presentation of the theory is organized as follows.

In this Section we present the formal syntax of our logic. This syntax is extremely simple and is not the syntax implemented in the theorem-prover. We then develop a large number of syntactic conventions used to describe the axioms and rules of inference.

In the next Section we present the axioms and the rules of inference.

Once we have completed the formal development of the logic we turn, in Section IMPLEMENTEDSYNTAX, to a description of the implemented syntax.

1.1. Syntax

The variables and function symbols of our language are taken from the set of "symbols" defined below.

A sequence of characters, s, is a symbol if and only if (i) s is nonempty, (ii) each character in s is a member of the set:

```
{A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 0 1 2 3 4 5 6 7 8 9
 -}
```

(iii) the first character of s is not a digit or hyphen.

Examples: PLUS, ADD1, X, and PRIME-FACTORS are symbols and thus are also variable symbols and function symbols. A/B, 123, and 1AB are not.

Associated with every function symbol is a nonnegative integer called the arity of the symbol. The arity indicates how many argument terms must follow each application of the function symbol. The arity of each primitive function symbol is given in the table below. We also include brief descriptive comments in the hopes that they will make subsequent examples more meaningful.

symbol	arity	comment
ADD1	1	successor function for natural numbers
ADD-TO-SET	2	adds an element to a list if not present
AND	2	logical and
APPEND	2	list concatenation

APPLY	2	application of function to arguments
ARITY	1	number of arguments expected by function
CAR	1	first component of ordered pair
CDR	1	second component of ordered pair
CONS	2	constructs ordered pairs
COUNT	1	size of a shell object
DIFFERENCE	2	difference of two natural numbers
EQUAL	2	equality predicate
FALSE	0	false object
FALSEP	1	predicate for recognizing FALSE
FIX	1	coerces argument to 0 if not numeric
FOR	6	general purpose quantifier
GEQ	2	greater than or equal on natural numbers
GREATERP	2	greater than on natural numbers
IF	3	if-then-else
IMPLIES	2	logical implication
INTERP	3	subroutine of INTERPRET
INTERPRET	2	evaluates tame forms in the logic
INTERPRET-LIST	2	evaluates a list of tame forms
LAST	1	last CDR of an object
LENGTH	1	length of a list
LEQ	2	less than or equal on natural numbers
LESSP	2	less than on natural numbers
LISTP	1	recognizes ordered pairs
LITATOM	1	recognizes literal atoms
LOOKUP	2	looks up value of atom in alist
MAX	2	maximum of two natural numbers
MEMBER	2	membership predicate
MINUS	1	constructs negative of a natural number
NEGATIVEP	1	recognizes negatives
NEGATIVE-GUTS	1	absolute value of a negative
NLISTP	1	negation of LISTP
NOT	1	logical negation
NUMBERP	1	recognizes natural numbers
OR	2	logical or
ORDINALP	1	recognizes ordinals
ORDP	1	a subfunction of ORDINALP
ORD-LESSP	2	less than on ordinals up to epsilon$_0$
PACK	1	constructs a literal atom from print name
PLUS	2	sum of two natural numbers
QUANTIFIER-INITIAL-VALUE	1	initial value of a quantifer
QUANTIFIER-OPERATION	3	operation performed by quantifier
QUOTIENT	2	natural quotient of two natural numbers
REMAINDER	2	mod
SUB1	1	predecessor function on natural numbers
TAME-FORMP	1	subroutine of INTERP
TIMES	2	product of two natural numbers
TRUE	0	true object
TRUEP	1	recognizes TRUE
SUBSETP	2	subset predicate
UNDEF	1	subroutine of INTERP
UNION	2	union of two lists
UNPACK	1	explodes litatom into print name
ZERO	0	0
ZEROP	1	recognizes 0 and non-natural numbers

The arity of each user-introduced function symbol is declared when the symbol is first used as a function symbol.

A term is either a variable symbol or else is a sequence consisting of a function symbol of arity n followed by n terms. We enclose non-variable terms in parentheses.

Examples: The following are terms:

```
(ZERO)

(ADD1 X)

(PLUS (ADD1 X) (ZERO))

(IF B
    (ZERO)
    (ADD1 X))
```

We present our axioms as formulas in the familiar syntax of propositional calculus with equality.

1.2. Syntactic Concepts

To talk about terms, it is convenient to use so-called "metavariables" that are understood by the reader to stand for certain variables, function symbols, or terms. In this document we use lower case words to denote metavariables.

Example: If f denotes the function symbol PLUS, and t denotes the term (ADD1 Y), then (f t X) denotes the term (PLUS (ADD1 Y) X).

If i is an integer, then by an abuse of notation we let Xi denote the variable whose first character is X and whose other characters are the decimal representation of i.

Example: If i is 4, Xi is the variable symbol X4.

A term t is a call of fn with arguments a_1, ..., a_n iff t has the form (fn a_1 ... a_n).

If a term t is a call of fn we say fn is the top function symbol of t. A function symbol fn is called in a term t iff either t is a call of fn or t is a nonvariable term and fn is called in an argument of t. The subterms of a term t is {t} if t is a variable symbol and otherwise is the union of {t} together with the union of the subterms of the arguments of t. The variables of a term t is the set of variable subterms of t.

Example: The term (PLUS X Y) is a call of PLUS with arguments X and Y. PLUS is called in (IF A (PLUS X Y) B). The set of subterms of (PLUS X Y) is {(PLUS X Y), X, Y}. The set of variables of (PLUS X Y) is {X Y}.

A finite set s of ordered pairs is said to be a substitution provided that for each ordered pair <v,t> in s, v is a variable, t is a term and no other member of s has v as its first component. The result of substituting a substitution s into a term p (denoted p/s) is the term obtained by simultaneously replacing, for each <v,t> in s, each occurrence of v as a variable in p with t. We sometimes say p/s is the result of instantiating p with s. We say that a term p' is an instance of p if there is a substitution s such that p' is p/s.

Example: If s is {<X,(ADD1 Y)> <Y,Z> <G,FOO>} then s is a substitution. If p is the term

```
(PLUS    X    (G Y    X))
```

then p/s is the term

```
(PLUS (ADD1 Y) (G Z (ADD1 Y))).
```

Note that even though the substitution contains the pair <G,FOO> the occurrence of G in p was not replaced by FOO since G does not occur as a variable in p.

We adopt the notational convention of sometimes writing a term where a formula is expected (e.g., we may refer to the "theorem" p, where p is a term). When we write a term p where a formula is expected, it is an abbreviation for the formula p≠(FALSE).

If a term p is a theorem, then by the rule of instantiation, the result of substituting any substitution into p is a theorem.

We use the symbols T and F as abbreviations for the terms (TRUE) and (FALSE), respectively. We do not use T and F as variable symbols.

We say term t is the nth CDR nest around the term x iff n is a natural number and either (i) n is 0 and t is x or (ii) n>0 and t is (CDR t') where t' is the n-1st CDR nest around x. When we write (CDR^n x) where a term is expected it is an abbreviation for the nth CDR nest around x.

Example: (CDR^2 A) is (CDR (CDR A)).

We say t is the fn nest around b for s iff t and b are terms, fn is a function symbol of arity 2, s is a finite sequence of terms, and either (i) s is empty and t is b or (ii) s is not empty and t is (fn t_1 t_2) where t_1 is the first element of s and t_2 is the fn nest around b for the remaining elements of s. When we write (fn t_1 ... t_n)@b where a term is expected it is an abbreviation for the fn nest around b for t_1, ..., t_n.

Examples: The OR nest around F for A, B, and C is the term (OR A (OR B (OR C F))), which may also be written (OR A B C)@F.

The basic axioms are the axioms and definitions in Section THEORY.

Formula t can be proved directly from a set of axioms A if and only if t may be derived from the axioms in A by applying the following rules of inference:

- the propositional calculus with equality and function symbols;
- the rule of inference that any instance of a theorem is a theorem; and
- the principle of induction as stated in subsection INDUCTION.

There are five kinds of axiomatic acts: (a) an application of the shell principle (subsection SHELLS), (b) an application of the principle of definition (subsection DEFNS), (c) an application of the reflection principle (subsection REFLECT), (d) the declaration of a "new" function symbol (subsection DCL), and (e) the addition of an arbitrary formula as an axiom.

Each such act adds a set of axioms. The axioms added by an application of the first four acts are described in the relevant subsections. The axioms added by the addition of an arbitrary formula is the singleton set consisting of the formula.

A history h is a finite sequence of axiomatic acts such that either (i) h is empty or (ii) h is obtained by concatenating to the end of a history h' an axiomatic act that is "admissible" under h'. An arbitrary axiom is admissible under any h'. The specification of the shell, definitional, reflection, and declaration principles define "admissiblity" in those instances.

The axioms of a history h is the union of the basic axioms together with the union of the axioms added by each act in h.

A function symbol fn is new in a history h iff fn is called in no axiom of h. A term t is old in a history h iff no function symbol called in t is new in h.

The axiomatic act of adding a shell, if admissible, adds a set of axioms that describe a "new" inductively constructed data type. Each application names a "constructor" function symbol, a "recognizer" function symbol, and some "accessors." In addition, the application may optionally name a "bottom" function symbol. To describe the admissibility criteria and the axioms added we make the following conventions.

The constructor function symbols of a history h is the union of {ADD1 CONS PACK MINUS} and the set of function symbols consisting exactly of the constructor function symbol of every application of the shell principle in h. The recognizer function symbols of a history h is union of {TRUEP FALSEP NUMBERP LISTP LITATOM NEGATIVEP} and the set consisting exactly of the recognizer function symbol of every application of the shell principle in h. The bottom function symbols of a history h is union of {TRUE, FALSE, ZERO} and the set consisting exactly of the bottom function symbol of every application of the shell principle in h for which a bottom function symbol was supplied.

We say r is the type of fn iff either (i) r is given as the type of fn in the table below or (ii) fn is a constructor or bottom function symbol introduced in the same axiomatic act in which r was the recognizer function symbol.

fn	type of fn
ADD1	NUMBERP
ZERO	NUMBERP
CONS	LISTP
PACK	LITATOM
MINUS	NEGATIVEP

A type restriction over a set of function symbols s is a pair <flg,s'> where flg is either the word ONE-OF or NONE-OF and s' is a finite sequence every element of which is an element of s.

A function symbol fn satisfies a type restriction <flg,s'> iff either flg is ONE-OF and fn is an element of s' or flg is NONE-OF and flg is not an element of s'.

We say t is the type restriction term for a type restriction <flg,(r_1 ... r_n)> and the variable symbol v iff flg is ONE-OF and t is (OR (r_1 v) ... (r_n v))@F or flg is NONE-OF and t is (NOT (OR (r_1 v) ... (r_n v))@F).

Examples: Let tr_1 be the pair <ONE-OF,(LISTP LITATOM)>. Then tr_1 is a type restriction over the set {NUMBERP LISTP LITATOM}. The function symbol LISTP satisfies tr_1 but the function symbol NUMBERP does not. The type restriction term for tr_1 and X1 is (OR (LISTP X1) (OR (LITATOM X1) F)). Let tr_2 be the pair <NONE-OF,(NUMBERP)>. Then tr_2 is a type restriction over the set {NUMBERP LISTP LITATOM}. The function symbol LISTP satisfies tr_2 but the function symbol NUMBERP does not. The type restriction term for tr_2 and X2 is (NOT (OR (NUMBERP X2) F)).

We say tr is the ith type restriction for a constructor function symbol fn iff $1 \leq i \leq n$ and either tr is as given by the table below or tr is the ith type restriction specified in the axiomatic act in which fn was introduced.

fn	1st type restriction	2nd type restriction (if applicable)
ADD1	<ONE.OF, (NUMBERP)>	
CONS	<NONE.OF, ()>	<NONE.OF, ()>
PACK	<NONE.OF, ()>	

```
MINUS          <ONE.OF, (NUMBERP)>
```

Below we give the *shell axioms for*:

```
constructor const of n arguments
with (optionally, bottom function btm)
recognizer r,
accessors ac_1, ..., ac_n,
type restrictions tr_1, ..., tr_n, and
default functions dv_1, ..., dv_n,
```

in history h, where const is a function symbol of n arguments, btm (if supplied) is a function symbol of no arguments, r and the ac_i are function symbols of 1 argument, the tr_i are type restrictions over the recognizers of h together with the symbol r, and the dv_i are function symbols of no arguments. In the formulas below, T should be used for all occurrences of (r (btm)) and F used for all terms of the form (EQUAL x (btm))), if no btm is supplied.

```
(1) (OR (EQUAL (r X) T)
        (EQUAL (r X) F)),

    (r (const X1 ... Xn)),

    (r (btm)),

    (NOT (EQUAL (const X1 ... Xn) (btm))), and

    (IMPLIES (AND (r X)
                  (NOT (EQUAL X (btm))))
             (EQUAL (const (ac_1 X) ... (ac_n X))
                    X));

(2) for each i from 1 to n, the following formula:

    (IMPLIES trt_i
             (EQUAL (ac_i (const X1 ... Xn))
                    Xi))
    where trt_i is the type restriction term for
    tr_i and Xi;

(3) for each i from 1 to n, the following formula:

    (IMPLIES (OR (NOT (r X))
                 (OR (EQUAL X (btm))
                     (AND (NOT trt_i)
                          (EQUAL X (const X1 ... Xn)))))
             (EQUAL (ac_i X) (dv_i)))
    where trt_i is the type restriction term for
    tr_i and Xi;

(4) the formulas:

    (NOT (r T)) and


    (NOT (r F));


(5) for each recognizer, r', in the recognizer functions
    of h the formula:
        (IMPLIES (r X) (NOT (r' X)));

    (6) the formula:
        (IMPLIES (r X)
                 (EQUAL (COUNT X)
                        (IF (EQUAL X (btm))
                            (ZERO)
                            (ADD1 (PLUS (ac_1 X)
                                        ...
                                        (ac_n X))@(ZERO)))))
```

We say t is an explicit value term in a history h iff t is a term and either (i) t is a call of a bottom function symbol in h, or (ii) t is a call of a constructor function symbol fn in h on arguments a_1, ..., a_n and for each i from 1 to n, a_i is an explicit value term in h and the type of the top function symbol of a_i satisfies the ith type restriction for the constructor function fn. We frequently omit reference to the history h when it is obvious by context.

Examples: The following are explicit value terms:

```
(ADD1 (ADD1 (ZERO)))

(CONS (PACK (ZERO)) (CONS (TRUE) (ADD1 (ZERO))))
```

The term (ADD1 X) is not an explicit value. The term (ADD1 (TRUE)) is not an explicit value, because the top function symbol of (TRUE) does not satisfy the type restriction, <ONE-OF, (NUMBERP)>, for the first argument of ADD1.

We next develop the notion that certain explicit value terms are the "quotations" of other terms. We begin by setting up the correspondence between the LITATOMs of the logic and the symbols of our syntax.

We say a term e is the NUMBERP corresponding to the natural number n iff either (i) n is 0 and e is (ZERO) or (ii) n is nonzero and e is (ADD1 e') where e' is the NUMBERP corresponding to n-1.

Example: The NUMBERP corresponding to 2 is (ADD1 (ADD1 (ZERO))).

When we write a nonnegative integer, n, where a term is expected, the integer is an abbreviation of the NUMBERP corresponding to n.

Example: The term (PLUS 2 X) is an abbreviation for (PLUS (ADD1 (ADD1 (ZERO))) X).

We say a term e is the explosion of a sequence of ASCII characters, s, iff either (i) s is empty and e is (ZERO) or (ii) s is a character c followed by some sequence s' and e is (CONS i e') where i is the NUMBERP corresponding to the ASCII code for c and e' is the explosion of s'.

Example: The ASCII codes for the characters A, B, and C are 65, 66, and 67 respectively. Then the explosion of ABC is:

```
(CONS 65 (CONS 66 (CONS 67 0))).
```

We say the term e is the LITATOM corresponding to a symbol s iff e is the term (PACK e') where e' is the explosion of s.

When we write a symbol s enclosed in quotation marks, e.g., "PLUS", where a term is expected, it abbreviates the LITATOM corresponding to s.

Example: When we write "ABC" where a term is expected we mean the LITATOM corresponding to ABC, i.e., the term

```
(PACK (CONS 65 (CONS 66 (CONS 67 0)))).
```

The use of the quotation mark convention is confined to the formal explication of the theory. In the implemented syntax we have a much more elaborate convention that permits the abbreviation of arbitrary explicit values.

We now define the notion of "quotation." We use LITATOMS to represent the variable and function symbols and LISTPs to stich the pieces together. However, we desire also to permit explicit values to be quoted in a special way. This makes the notion of "quotation" depend upon the notion of "explicit value," which, recall, involves a particular history h from which the constructor and bottom functions are drawn. This is the only sense in which the notion of "quotation" depends upon a history.

We say e is a quotation of t (in some history h which is implicit throughout this definition) iff e and t are terms and either (i) t is a variable symbol and e is the LITATOM corresponding to t, (ii) t is an explicit value term and e is (CONS "QUOTE" (CONS t "NIL")), or (iii) t has the form (fn a_1 ... a_n) and e is (CONS efn elst) where efn is the LITATOM corresponding to fn and elst is a "quotation list" (see below) of a_1 ... a_n.

We say elst is a quotation list of tlst (in some history h which is implicit throughout this definition) iff elst is a term and tlst is a sequence of terms, and either (i) tlst is empty and elst is "NIL" or (ii) tlst consists of a term t followed by a sequence tlst' and elst is (CONS e elst') where e is a quotation of t and elst' is a quotation list of tlst'.

Examples: Below we give some terms and examples of their quotations.

term	quotation
`ABC`	`"ABC"`
`(ZERO)`	`(CONS "ZERO" "NIL")`
`(ZERO)`	`(CONS "QUOTE" (CONS (ZERO) "NIL"))`
`(ADD1 X)`	`(CONS "ADD1" (CONS "X" "NIL"))`

The meta axioms for f, where f is a function symbol of arity n, are given below. In the formulas we use "f" as a metavariable denoting the LITATOM corresponding to f and nn as a metavariable denoting the NUMBERP corresponding to n.

```
(EQUAL (APPLY "f" L)
       (f (CAR (CDR^0 L)) ... (CAR (CDR^(n-1) L))))

(EQUAL (ARITY "f") nn)
```

A term t is tame (in some history h which is implicit throughout this definition) iff either (i) t is a variable, or (ii) t is a call of a function symbol fn on arguments a_1, ..., a_n, each a_i is tame, and one of the following obtains:

- fn is INTERPRET and a_1 is a quotation of a term t_1, t_1 is old in h, and t_1 is tame; or
- fn is INTERPRET-LIST and a_1 is a quotation list of a sequence of terms $t_{1,1}$, ..., $t_{1,k}$, each $t_{1,i}$ is old in h, and each $t_{1,i}$ is tame; or

- fn is INTERP, a_1 is an explicit value and either (i) a_1 is not "LIST" and a_2 is a quotation of a term t_2, t_2 is old in h and t_2 is tame, or (ii) a_1 is "LIST" and a_2 is a quotation list of a sequence of terms $t_{2,1}$, ..., $t_{2,k}$, each $t_{2,i}$ is old in h and each $t_{2,i}$ is tame; or
- fn is FOR, a_3 is a quotation of a term t_3, t_3 is old in h, and t_3 is tame, and a_5 is a quotation of a term t_5, t_5 is old in h and t_5 is tame; or
- fn is not INTERPRET, INTERPRET-LIST, INTERP, FOR, or APPLY.

Note that any term not calling INTERP, INTERPRET, INTERPRET-LIST, APPLY or FOR is tame. Furthermore, if the only function symbol called among those just listed is FOR, then the term is tame provided only that the third and fifth arguments of the FOR are quotations of old, tame terms.

Examples: The following terms are tame:

```
X

(ADD1 X)

(INTERPRET (CONS "ADD1" (CONS "X" "NIL"))
           A)

(INTERP "LIST"
        (CONS (CONS "ADD1" (CONS "X" "NIL"))
              (CONS "Y"
                    "NIL"))
        A)
```

The last two examples may be displayed in the implemented syntax (as opposed to the simple syntax) as follows:

```
(INTERPRET '(ADD1 X) A)

(INTERP 'LIST '((ADD1 X) Y) A)
```

The following term, displayed in the implemented syntax, is tame

```
(INTERPRET '(INTERPRET '(ADD1 X) A) B),
```

even though the interpreted form involves INTERPRET. The term (INTERPRET (CONS FN ARGS) A) is not tame because (CONS FN ARGS) is not the quotation of a term.

A term t contains a hidden call of a function symbol fn (in some history h which is implicit throughout this definition) iff t is a call of a function symbol fn on arguments a_1, ..., a_n and either one of the a_i contains a hidden call of fn or one of the following obtains:

- fn is INTERPRET and a_1 is a quotation of a term t_1 and t_1 either calls fn or contains a hidden call of fn;
- fn is INTERPRET-LIST and a_1 is a quotation list of a sequence of terms $t_{1,1}$, ..., $t_{1,k}$ and some $t_{1,i}$ either calls fn or contains a hidden call of fn;
- fn is INTERP, a_1 is an explicit value and either (i) a_1 is not "LIST" and a_2 is a quotation of a term t_2 and t_2 either calls fn or contains a hidden call of fn, or (ii) a_1 is "LIST" and a_2 is a quotation list of a sequence of terms $t_{2,1}$, ..., $t_{2,k}$ and some $t_{2,i}$ either calls fn or contains a hidden call of fn;
- fn is FOR and either (i) a_3 is a quotation of a term t_3 and t_3 either calls fn or contains a hidden call of fn or (ii) a_5 is a quotation of a term t_5 and t_5 either calls fn or contains a hidden call of fn;

Examples: Suppose FN is a function symbol of 1 argument. Then the term (ADD1 (FN X)) calls FN but contains no hidden calls of FN. The term (ADD1 (INTERPRET (CONS "FN" (CONS "X" "NIL")) A)) does not call FN but does contain a hidden call of FN.

We say that a term t governs an occurrence of a term s in a term b iff either b contains a subterm of the form (IF t p q) and the occurrence of s is in p, or if b contains a subterm of the form (IF t' p q), where t is (NOT t') and the occurrence of s is in q.

Examples: The terms P and (NOT Q) govern the first occurrence of S in:

```
(IF P
    (IF (IF Q A S)
        S
        B)
    C)
```

The terms P and (IF Q A S) govern the second occurrence of S.

2. The Formal Theory

We now present the axioms and rules of inference of our logic.

The axioms presented in the format:

```
Defining Axiom.
```
$(f\ x_1\ \ldots\ x_n)$ = body

have the special property that it can be shown (in a suitable theory of sets) that one and only one function f satisfies the equation.

In general we use the principle of definition

```
Definition.
```
$(f\ x_1\ \ldots\ x_n)$ = body

to add such axioms. However, the admissibility requirements on the principle of definition require that certain theorems be provable -- theorems that in fact guarantee that one and only one function satisfies the equation. However, until enough of the logic has been built up, the required theorems cannot be proved.

Thus, the presentation of the logic is structured as follows. First we list a collection of axioms defining many of the most primitive function symbols. Then we present the induction principle and the extension principles, including the definitional principle. Then we invoke the definitional principle to add the definitions of many useful functions.

2.1. TRUE, FALSE, IF and EQUAL

```
Axiom.
T ≠ F

Axiom.
X = Y -> (EQUAL X Y) = T

Axiom.
X ≠ Y -> (EQUAL X Y) = F
```

```
Axiom.
X = F -> (IF X Y Z) = Z

Axiom.
X ≠ F -> (IF X Y Z) = Y.


Defining Axiom.
(TRUEP X) = (EQUAL X T)


Defining Axiom.
(FALSEP X) = (EQUAL X F)


Defining Axiom.
(NOT P)
   =
(IF P F T)

Defining Axiom.
(AND P Q)
   =
(IF P (IF Q T F) F)

Defining Axiom.
(OR P Q)
   =
(IF P T (IF Q T F))

Defining Axiom.
(IMPLIES P Q)
   =
(IF P (IF Q T F) T).
```

2.2. Natural Numbers

We assume the shell axioms for

```
constructor ADD1 of one argument
with bottom object ZERO,
recognizer NUMBERP,
accessor SUB1,
type restriction <ONE-OF,(NUMBERP)>,
default function ZERO.
```

We now add three additional axioms about COUNT.

```
Axiom.
(NUMBERP (COUNT X))

Axiom.
(EQUAL (COUNT T) 0)

Axiom.
(EQUAL (COUNT F) 0)
```

We now introduce the axiom defining PLUS, which was used in the shell axioms.

```
Defining Axiom.
(ZEROP X)
  =
(OR (EQUAL X 0) (NOT (NUMBERP X)))
```

```
Defining Axiom.
(FIX X) = (IF (NUMBERP X) X 0)
```

```
Defining Axiom.
(PLUS X Y)
  =
(IF (ZEROP X)
    (FIX Y)
    (ADD1 (PLUS (SUB1 X) Y)))
```

2.3. Ordered Pairs

We assume the shell axioms for

```
constructor CONS of two arguments
with recognizer LISTP,
accessors CAR and CDR,
default functions ZERO and ZERO.
```

2.4. Literal Atoms

We assume the shell axioms for:

```
constructor PACK of one argument
recognizer LITATOM,
accessor UNPACK,
default function ZERO.
```

2.5. Negative Integers

We assume the shell axioms for

```
constructor MINUS of one argument
with recognizer NEGATIVEP,
accessor NEGATIVE.GUTS,
type restriction <ONE-OF,(NUMBERP)>,
default function ZERO.
```

2.6. Ordinals

We now use NUMBERPs and LISTPs to represent the ordinals up to epsilon$_0$. The table below illustrates our representation. The notation used is that of the implemented syntax, not the formal syntax.

ordinal	representation
0	0
1	1
2	2
3	3
...	...
w	'(1 . 0)

$\omega+1$	'(1 . 1)
$\omega+2$	'(1 . 2)
...	...
2ω	'(1 1 . 0)
$2\omega+1$	'(1 1 . 1)
...	...
...	...
ω^2	'(2 . 0)
...	...
$\omega^2+\omega+3$	'(2 1 . 3)
...	...
ω^3	'(3 . 0)
...	...
...	...
ω^ω	'((1 . 0) . 0)
...	...

We assume the following axiom defining LESSP, the less than relation on the natural numbers.

```
Defining Axiom.
(LESSP X Y)
   =
(IF (ZEROP Y)
    F
    (IF (ZEROP X)
        T
        (LESSP (SUB1 X) (SUB1 Y))))
```

The less than relation on the ordinals is then defined as follows:

```
Defining Axiom.
(ORD-LESSP X Y)
   =
(IF (NLISTP X)
    (IF (NLISTP Y)
        (LESSP X Y)
        T)
    (IF (NLISTP Y)
        F
        (IF (ORD-LESSP (CAR X) (CAR Y))
            T
            (AND (EQUAL (CAR X) (CAR Y))
                 (ORD-LESSP (CDR X) (CDR Y))))))
```

The function for recognizing ordinals is defined as follows:

```
Defining Axiom.
(ORDINALP X) = (OR (EQUAL X 0) (ORDP X))
```

where

```
Defining Axiom.
(ORDP X)
   =
(IF (NLISTP X)
    (IF (NUMBERP X) (NOT (EQUAL X 0)) F)
    (AND (ORDP (CAR X))
         (ORDP (CDR X))
         (IF (LISTP (CDR X))
             (NOT (ORD-LESSP (CAR X) (CADR X)))
             T)))
```

Our principles of induction and definition are based on the assumption that ORD-LESSP is well-founded on the ORDINALPs. That is, there is no infinite sequence o_0, o_1, o_2, ... with the property that for each natural number i, (ORDINALP o_i) and (ORD-LESSP o_{i+1} o_i).

The well-founded lexicographic relation on n-tuples of natural numbers induced by LESSP can be obtained by an appropriate use of ORD-LESSP. For example, suppose if i_1, j_1, i_2, and j_2 are all NUMBERPs. Then the pair $<i_1,j_1>$ is lexicographically smaller than $<i_2,j_2>$ precisely when

```
(ORD-LESSP (CONS (CONS i1 0) j1)
           (CONS (CONS i2 0) j2)).
```

2.7. Basic Meta Axioms

In general, every history will include the meta axioms for every function symbol used in the axioms of the history with the exceptions of APPLY and UNDEF. The axiomatic acts (with which the user constructs histories) are defined to add the appropriate meta axioms for each new function symbol. However, in constructing the basic axioms we have not yet begun to use the axiomatic acts and thus have to assume the meta axioms explicitly for each function symbol introduced so far.

We assume the meta axioms for the function symbols TRUE, FALSE, IF, EQUAL, TRUEP, FALSEP, NOT, AND, OR, IMPLIES, COUNT, ADD1, ZERO, NUMBERP, SUB1, ZEROP, FIX, PLUS, CONS, LISTP, CAR, CDR, PACK, LITATOM, UNPACK, MINUS, NEGATIVEP, NEGATIVE-GUTS, LESSP, ORD-LESSP, ORDP, ORDINALP and ARITY.

We assume the following axioms:

```
(EQUAL (ARITY "APPLY") 2)

(EQUAL (ARITY "UNDEF") 1)
```

2.8. Induction

The rules of inference of our logic consist of the usual rules of inference of propositional calculus with equality, including the rule that any instance of a theorem is a theorem, together with the following principle of mathematical induction:

```
Suppose:

    (a) p is a term;
    (b) m is a function symbol of n arguments;
    (c) x1, ..., xn are distinct variables;
    (d) q1, ..., qk are terms;
    (e) h1, ..., hk are positive integers;
    (f) it is a theorem that (ORDINALP (m x1 ... xn));
        and
    (g) for 1≤i≤k and 1≤j≤hi, si,j
        is a substitution and it is a theorem that:

        (IMPLIES qi (ORD-LESSP (m x1 ... xn)/si,j
                               (m x1 ... xn))).

Then p is a theorem if

        (IMPLIES (AND (NOT q1) ... (NOT qk))@T
                 p)
```

is a theorem and

for each $1 \le i \le k$,

```
    (IMPLIES (AND q_i p/s_{i,1} ... p/s_{i,h_i})@T
             p)
```

is a theorem.

2.9. The Shell Principle

The axiomatic act:

> <u>Shell</u> <u>Definition</u>.
> <u>add</u> <u>the</u> <u>shell</u> const <u>of</u> n <u>arguments</u>
> <u>with</u> (<u>optionally</u>, <u>bottom</u> <u>function</u> btm)
> <u>recognizer</u> r,
> <u>accessors</u> ac_1, ..., ac_n,
> <u>type</u> <u>restrictions</u> tr_1, ..., tr_n, <u>and</u>
> <u>default</u> <u>functions</u> dv_1, ..., dv_n,

is admissible under the history h provided:

(a) const is a new function symbol of n arguments, (btm is a new function symbol of no arguments, if a bottom object is supplied), r, ac_1, ..., ac_n are new function symbols of one argument, and all the above function symbols are distinct;

(b) each tr_i is a type restriction over the recognizers of h together with the symbol r;

(c) for each i, dv_i is either btm or one of the bottom functions of h; and

(d) for each i, if dv_i is btm then r satisfies tr_i and otherwise the type of dv_i satisfies tr_i.

If admissible we add the shell axioms for

> constructor const of n arguments
> with (optionally, bottom function btm)
> recognizer r,
> accessors ac_1, ..., ac_n,
> type restrictions tr_1, ..., tr_n, and
> default functions dv_1, ..., dv_n,

along with the meta axioms for const, r, ac_1, ..., ac_n, and (if btm was supplied) btm.

If the tr_i are not specified, they should each be assumed to be <NONE-OF,()>.

2.10. The Principle of Definition

The axiomatic act:

<u>Definition</u>. (f x_1 ... x_n) = body

is admissible under the history h provided:

(a) f is a function symbol of n arguments and is new in h;

(b) x_1, ..., x_n are distinct variables;

(c) body is a term and mentions no symbol as a variable other than x_1, ..., x_n;

(d) body is a tame term in history h;

(e) body contains no hidden calls of f in history h; and

(f) there is a function symbol m of n arguments, such that (i) (ORDINALP (m x_1 ... x_n)) can be proved directly in h, and (ii) for each occurrence of a subterm of the form (f y_1 ... y_n) in body and the terms t_1, ..., t_k governing it, the following formula can be proved directly in h:

```
(IMPLIES (AND t_1 ... t_k)@T
         (ORD-LESSP (m y_1 ... y_n)
                    (m x_1 ... x_n))).
```

If admissible, we add the meta axioms for f and the axiom:

(f x_1 ... x_n) = body.

2.11. The Principle of Reflection

The axiomatic act:

<u>Reflect</u>. (f x_1 ... x_n) = body

is admissible under the history h provided:

(a) f is a function symbol of n arguments and is new in h;

(b) x_1, ..., x_n are distinct variables;

(c) body is a term and mentions no symbol as a variable other than x_1, ..., x_n;

(d) body is a tame term in history h;

(e) body contains no hidden calls of f in history h;

(f) there is a function symbol m of n arguments, a function symbol f' of n arguments, and a term body' obtained by replacing every occurrence of f

```
as a function symbol in body by f', such that
(i) (ORDINALP (m x1 ... xn)) can be
proved directly in h, (ii) the formula
(EQUAL (f' x1 ... xn) body')
can be proved directly in h and (iii) for each
occurrence of a subterm of the form (f' y1 ... yn)
in body' and the terms t'1, ..., t'k governing it,
the following formula can be proved directly in h:

(IMPLIES (AND t'1 ... t'k)@T
         (ORD-LESSP (m y1 ... yn)
                    (m x1 ... xn))).
```

If admissible, we add the meta axioms for f and the axiom:

(f x_1 ... x_n) = body.

2.12. The Principle of Declaration

The axiomatic act:

<u>Declare</u>. (f x_1 ... x_n).

is admissible in history h provided that:

(a) f is a function symbol of n arguments and is new in h; and

(b) x_1, ..., x_n are distinct variable symbols.

The axioms added by an admissible declaration are the meta axioms for f.

2.13. Useful Function Definitions

We now introduce a variety of useful functions. These functions are part of the basic theory either because (i) they are used in our implementation of the interpreter (e.g., LOOKUP), (ii) we have found it necessary, from a practical point of view, to build knowledge of them into the theorem-prover (e.g., DIFFERENCE is used in the linear arithmetic decision procedure), or (iii) the von Neumann machine on which the theorem-prover runs provides means of computing the functions that are significantly faster than merely compiling the recursive definitions (e.g., QUOTIENT). Our interest in computational efficiency stems from our desire that the logic be a useful functional programming language and not from theorem-proving considerations. Each of the following functions is introduced with the principle of definition and hence for each we also assume the corresponding meta axioms.

We first define some useful functions on the natural numbers.

```
Definition.
(GREATERP X Y) = (LESSP Y X)

Definition.
(LEQ X Y) = (NOT (LESSP Y X))

Definition.
(GEQ X Y) = (NOT (LESSP X Y))

Definition.
(MAX X Y) = (IF (LESSP X Y) Y (FIX X))
```

```
Definition.
(DIFFERENCE I J)
  =
(IF (ZEROP I)
    0
    (IF (ZEROP J)
        I
        (DIFFERENCE (SUB1 I) (SUB1 J))))

Definition.
(TIMES I J)
  =
(IF (ZEROP I)
    0
    (PLUS J (TIMES (SUB1 I) J)))

Definition.
(QUOTIENT I J)
  =
(IF (ZEROP J)
    0
    (IF (LESSP I J)
        0
        (ADD1 (QUOTIENT (DIFFERENCE I J) J))))

Definition.
(REMAINDER I J)
  =
(IF (ZEROP J)
    (FIX I)
    (IF (LESSP I J)
        (FIX I)
        (REMAINDER (DIFFERENCE I J) J)))
```

Next we define some useful list processing functions.

```
Definition.
(NLISTP X) = (NOT (LISTP X))

Definition.
(LENGTH LST)
  =
(IF (LISTP LST)
    (ADD1 (LENGTH (CDR LST)))
    0)

Definition.
(MEMBER X LST)
  =
(IF (NLISTP LST)
    F
    (IF (EQUAL X (CAR LST))
        T
        (MEMBER X (CDR LST))))

Definition.
(UNION X Y)
  =
(IF (LISTP X)
    (IF (MEMBER (CAR X) Y)
        (UNION (CDR X) Y)
        (CONS (CAR X) (UNION (CDR X) Y)))
    Y)
```

```
Definition.
(SUBSETP X Y)
  =
(IF (NLISTP X)
    T
    (IF (MEMBER (CAR X) Y)
        (SUBSETP (CDR X) Y)
        F))
```

```
Definition.
(ADD-TO-SET X SET)
  =
(IF (MEMBER X SET)
    SET
    (CONS X SET))
```

```
Definition.
(APPEND X Y)
  =
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y)
```

```
Definition.
(LAST L)
  =
(IF (LISTP L)
    (IF (LISTP (CDR L))
        (LAST (CDR L))
        L)
    L)
```

```
Definition.
(LOOKUP X ALIST)
  =
(IF (NLISTP ALIST)
    0
    (IF (AND (LISTP (CAR ALIST))
             (EQUAL X (CAR (CAR ALIST))))
        (CDR (CAR ALIST))
        (LOOKUP X (CDR ALIST))))
```

2.14. The Interpreter

```
Definition.
(TAME-FORMP X)
  =
(IF (NLISTP X)
    T
(IF (EQUAL (CAR X) "INTERP")
    (AND (LISTP (CAR (CDR^2 X)))
         (EQUAL (CAR (CAR (CDR^2 X))) "QUOTE"))
(IF (EQUAL (CAR X) "INTERPRET")
    (AND (LISTP (CAR (CDR X)))
         (EQUAL (CAR (CAR (CDR X))) "QUOTE"))
(IF (EQUAL (CAR X) "INTERPRET-LIST")
    (AND (LISTP (CAR (CDR X)))
         (EQUAL (CAR (CAR (CDR X))) "QUOTE"))
(IF (EQUAL (CAR X) "FOR")
    (AND (LISTP (CAR (CDR^3 X)))
         (EQUAL (CAR (CAR (CDR^3 X))) "QUOTE"))
    (NOT (EQUAL (CAR X) "APPLY")))))))
```

The following three axioms are inadmissible under the principle of definition because the bodies are not tame. However, if UNDEF is defined as an arbitrary function of one argument and APPLY is defined to satisfy all of the meta axioms in a history (which is always possible), it can be shown that there exists one and only one function satisfying each of the following equations.

```
Defining Axiom.
(INTERP FLG FORM ALIST)
  =
(IF (EQUAL FLG "LIST")
    (IF (NLISTP FORM)
        "NIL"
        (CONS (INTERP T (CAR FORM) ALIST)
              (INTERP FLG (CDR FORM) ALIST)))
    (IF (LITATOM FORM)
        (LOOKUP FORM ALIST)
        (IF (NLISTP FORM)
            FORM
            (IF (EQUAL (CAR FORM) "QUOTE")
                (CAR (CDR FORM))
                (IF (TAME-FORMP FORM)
                    (APPLY (CAR FORM)
                           (INTERP "LIST" (CDR FORM) ALIST))
                    (UNDEF (CONS FORM (CONS ALIST "NIL"))))))))

Defining Axiom.
(INTERPRET FORM ALIST) = (INTERP T FORM ALIST)

Defining Axiom.
(INTERPRET-LIST LST ALIST) = (INTERP "LIST" LST ALIST)
```

We assume the meta axioms for INTERP, INTERPRET, and INTERPRET-LIST.

2.15. Quantification

We now define our general purpose quantifier function FOR.

```
Definition.
(QUANTIFIER-INITIAL-VALUE FN)
  =
(IF (EQUAL FN "ADD-TO-SET")  "NIL"
(IF (EQUAL FN "ALWAYS")      T
(IF (EQUAL FN "APPEND")      "NIL"
(IF (EQUAL FN "COLLECT")     "NIL"
(IF (EQUAL FN "COUNT")       0
(IF (EQUAL FN "DO-RETURN")   "NIL"
(IF (EQUAL FN "EXISTS")      F
(IF (EQUAL FN "MAX")         0
(IF (EQUAL FN "MULTIPLY")    1
(IF (EQUAL FN "SUM")         0
(IF (EQUAL FN "UNION")       "NIL"
                             0)))))))))))
```

```
Definition.
(QUANTIFIER-OPERATION FN ARG REST)
  =
(IF (EQUAL FN "ADD-TO-SET") (ADD-TO-SET ARG REST)
(IF (EQUAL FN "APPEND")     (APPEND ARG REST)
(IF (EQUAL FN "COLLECT")    (CONS ARG REST)
(IF (EQUAL FN "COUNT")      (IF ARG (ADD1 REST) REST)
(IF (EQUAL FN "DO-RETURN")  ARG
(IF (EQUAL FN "EXISTS")     (OR ARG REST)
(IF (EQUAL FN "MAX")        (MAX ARG REST)
(IF (EQUAL FN "MULTIPLY")   (TIMES ARG REST)
(IF (EQUAL FN "SUM")        (PLUS ARG REST)
(IF (EQUAL FN "UNION")      (UNION ARG REST)
                            0))))))))))
```

The following axiom is inadmissible under the principle of definition because the body is not tame. However, under the same conditions on UNDEF and APPLY discussed above, it can be shown that one and only one function satisfies the axiom.

```
Defining Axiom.
(FOR V L COND OP BODY A)
  =
(IF (NLISTP L)
    (QUANTIFIER-INITIAL-VALUE OP)
    (IF (INTERPRET COND (CONS (CONS V (CAR L)) A))
        (QUANTIFIER-OPERATION OP
           (INTERPRET BODY (CONS (CONS V (CAR L)) A))
           (FOR V (CDR L) COND OP BODY A))
        (FOR V (CDR L) COND OP BODY A)))
```

We assume the meta axioms for FOR.

3. The Implemented Syntax

The syntax used in the theorem-prover is somewhat more elaborate than the formal syntax. Every formula in the implemented syntax abbreviates a formula in the formal syntax, according to conventions described in this Section. However, not every formula in the formal syntax can be so abbreviated. For example, in the formal syntax T is a variable symbol, while in the implemented syntax T abbreviates the term (TRUE) and there is no way to write a reference to the variable symbol T.

The implemented syntax is complicated by three factors. First, we use the LISP reader to parse user typein into terms and thus must describe the lexical analysis performed by that system. This includes handling of the read macro character '. Second, we adopt the LISP "quote" notation for certain explicit values, including "dot" notation for LISTP constants. Third, we provide a variety of abbreviation conventions which are context sensitive in the sense that they are not applied to "quoted" constants. For example, while 1 is usually thought of as an abbreviation of (ADD1 (ZERO)) not all occurrences of 1 can be so replaced. In particular, (QUOTE 1) and (QUOTE (ADD1 (ZERO))) denote two different constants (the first is a NUMBERP and the second is a LISTP).

In order to define our syntax precisely we exhibit the lexical analyzer, parser, and term recognizer as functions defined in the formal logic itself. We ultimately define the function ISYNTAX which takes as input a list of numeric character codes and delivers either F or the quotation of a formal term. Suppose stream is the CONS nest around "NIL" of the ASCII character codes of the characters in the sequence s. If (ISYNTAX stream) is F, then s is an ill-formed expression in the implemented syntax. Otherwise (ISYNTAX stream) is the quotation of a formal term t and s is a well-formed expression in the implemented syntax and denotes the term t.

3.1. Examples of the Implemented Syntax

In this section we illustrate the implemented syntax by exhibiting some theorems about ISYNTAX. To make the examples more succinct, we first introduce some notational conventions.

We use the notation |s|, where s is sequence of ASCII characters other than the vertical bar character, to denote the CONS nest around "NIL" of the sequence of ASCII character codes for the successive characters in s.

Some of the relevant ASCII codes are:

char	code	char	code
<space>	32	0	48
'	39	1	49
(	40	...	...
)	41	9	57
+	43	A	65
-	45	B	66
.	46	...	...
		Z	90

For example, |(A. 1'| is an abbreviation for

```
(CONS 40 (CONS 65 (CONS 46 (CONS 32 (CONS 49 (CONS 39 "NIL"))))))
```

We say e is <u>the</u> <u>primitive</u> <u>quotation</u> of t iff e and t are terms and either (i) t is a variable symbol and e is the LITATOM corresponding to t or (ii) t has the form (fn a_1 ... a_n) and e is (CONS efn elst) where efn is the LITATOM corresponding to fn and elst is the CONS nest around "NIL" of the primitive quotations of each of the a_i. We will sometimes write ⌊t⌋ to mean the primitive quotation of t.

Thus, ⌊A⌋, the primitive quotation of the variable symbol A, is "A" or, equivalently, (PACK (CONS 65 0)). ⌊(CAR A)⌋ is (CONS "CAR" (CONS "A" "NIL")).

The primitive quotation of explicit value terms are cumbersomely large. For example, ⌊0⌋ is (CONS "ZERO" "NIL") and ⌊1⌋ is (CONS "ADD1" (CONS (CONS "ZERO" "NIL") "NIL")). ⌊"A"⌋ is shown below:

```
(CONS "PACK"
      (CONS (CONS "CONS" (CONS [65] (CONS [0] "NIL")))
            "NIL"))
```

We now illustrate the implemented syntax. Informally speaking, the expression (CONS T NIL) in the implemented syntax abbreviates the formal term (CONS (TRUE) "NIL"). This statement can be made precise as follows:

```
Theorem.
(EQUAL (ISYNTAX |(CONS T NIL)|) [(CONS (TRUE) "NIL")]).
```

In general, if iterm is a string of ASCII characters and term is a formal term, and (EQUAL (ISYNTAX |iterm|) ⌊term⌋) is a theorem, then we say iterm is an <u>implementation</u> <u>term</u> that <u>abbreviates</u> term.

Each of the iterms below abbreviates (CONS (TRUE) "NIL"):

```
iterm
```

```
    ( CONS    T       NIL )

    (CONS T
          NIL)

    (CONS
T NIL
)
    (CONS T    ;this is a comment
               ;and so is this.
          NIL  ;and here is another
          )
```

Here are some other example of iterms and the terms they abbreviate.

```
     iterm                        term

    (CONS 2 -1)            (CONS (ADD1 (ADD1 (ZERO)))
                                 (MINUS (ADD1 (ZERO))))

    (LIST A B C)           (CONS A (CONS B (CONS C "NIL"))).

    (CADDR X)              (CAR (CDR (CDR X))).

    (TIMES I J K)          (TIMES I (TIMES J K)).

    (QUOTE (A 1 C))        (CONS "A"
                                 (CONS (ADD1 (ZERO))
                                       (CONS "C" "NIL")))
    (QUOTE (A 1 . 0))      (CONS "A"
                                 (CONS (ADD1 (ZERO))
                                       (ZERO)))

    (QUOTE (T *1*TRUE))  (CONS "T"
                                 (CONS T "NIL"))

    (QUOTE (ADD1 X))     (CONS "ADD1" (CONS "X" "NIL"))
```

The last example illustrates one of the convenient aspects of the implementation syntax. If t is a formal term then (QUOTE t) is an implementation term that abbreviates a quotation of t.

The "single gritch" character, ', can be used to embed the following well-formed expression in a QUOTE. Thus:

```
    iterm                   term

    'A                      "A"

    '(ADD1 X)               (CONS "ADD1" (CONS "X" "NIL"))

    '(A 'B))                (CONS "A"
                                  (CONS (CONS "QUOTE"
                                              (CONS "B" "NIL"))
                                        "NIL"))
```

Our implementation of the QUOTE convention has special provisions for the inclusion of literal atoms that do not correspond to symbols (e.g., (PACK (CONS 64 0))) and the inclusion of user introduced shell constants. Such constants are written down using the special token *1*QUOTE.

For example, the following implementation term

```
(QUOTE (C B A (*1*QUOTE PACK (64 . 0))))
```

abbreviates the same formal term as

```
(CONS (PACK (CONS 67 0))
      (CONS (PACK (CONS 66 0))
            (CONS (PACK (CONS 65 0))
                  (CONS (PACK (CONS 64 0))
                        NIL)))).
```

In order to further illustrate use of *1*QUOTE, let us extend the initial history with the axiomatic act:

```
Shell Definition.
Add the shell PUSH of 2 arguments
with bottom function EMPTY-STACK,
recognizer STACKP,
accessors TOP and POP,
type restrictions <ONE-OF,(NUMBERP)> and <ONE-OF,(STACKP)>
and default functions ZERO and EMPTY-STACK.
```

Then the implemented term:

```
(LIST 'A (EMPTY-STACK)
      'B (PUSH 2 (EMPTY-STACK)))
```

represents the same formal term as represented by the implemented term:

```
'(A (*1*QUOTE EMPTY-STACK)
  B (*1*QUOTE PUSH 2 (*1*QUOTE EMPTY-STACK)))
```

However, *use of *1*QUOTE is restricted* so that it cannot be used to represent explicit values that could be written down inside QUOTE without use of *1*QUOTE. Thus, '((*1*QUOTE ZERO)) is ill-formed. In addition, *1*QUOTE cannot be used to write down terms that are not explicit values, e.g., '((*1*QUOTE PUSH 2 3)) is ill-formed because NUMBERP, the type of 3, does not satisfy the type restriction on the second argument of PUSH.

Finally, our implemented syntax contains an elaborate mechanism for the abbreviation of FOR expressions.

In the implementation syntax one can either write a 6 argument application of FOR – in which case each of the 6 arguments is simply translated – or one can write a 5 or 7 argument application. In the latter cases, certain of the "arguments" are "noise" words and others are treated as implementation terms which are translated and then embedded in QUOTEs. In addition, when a 5 or 7 argument FOR is used the translation routine automatically computes the association list used to assign values to the "free" variables occuring in the expression. For example the implementation term

```
(FOR X IN L WHEN (LESSP X 100) SUM (TIMES A B X))
```

abbreviates same term as:

```
(FOR 'X L '(LESSP X (QUOTE 100)) 'SUM '(TIMES A (TIMES B X))
     (LIST (CONS 'A A)
           (CONS 'B B)))
```

3.2. Some Preliminary Conventions

In the next three subsections we define formally the lexical analyzer, parser, and translator. Because of the need to determine whether a given term is an explicit value, we must provide functions that answer such questions as "is this the name of a shell constructor function?" and "what are the type restrictions on this shell?"

We use three such functions.

1. SHELL-BTM-TYPE: If X is the quotation of the bottom function symbol of the shell class with recognizer function symbol r, (SHELL-BTM-TYPE X) is equal to the quotation of r; otherwise (SHELL-BTM-TYPE X) is equal to F.
2. SHELL-CONS-TYPE: If X is the quotation of the constructor function symbol of the shell class with recognizer function symbol r, (SHELL-CONS-TYPE X) is equal to the quotation of r; otherwise (SHELL-BTM-TYPE X) is equal to F.
3. SHELL-CONS-TYPES: If X is not the quotation of the constructor function symbol of some shell class, (SHELL-CONS-TYPES X) is F. Otherwise, X is the quotation of some constructor function with type restrictions $<flg_1,s_1>$, ..., $<flg_n,s_n>$. Let tr_i be the CONS nest around "NIL" of the LITATOMs corresponding to flg_i and each of the symbols in s_i. Then (SHELL-CONS-TYPES X) is equal to the CONS list around "NIL" of tr_1, ..., tr_n.

Each of these functions could be defined for a given history. For example, in the empty history:

```
(SHELL-BTM-TYPE X)
   =
(IF (EQUAL X "TRUE") "TRUEP"
(IF (EQUAL X "FALSE") "FALSEP"
(IF (EQUAL X "ZERO") "NUMBERP"
    F)))

(SHELL-CONS-TYPE X)
   =
(IF (EQUAL X "ADD1") "NUMBERP"
(IF (EQUAL X "CONS") "LISTP"
(IF (EQUAL X "PACK") "LITATOM"
(IF (EQUAL X "MINUS") "NEGATIVEP"
    F))))

(SHELL-CONS-TYPES X)
   =
(IF (OR (EQUAL X "ADD1")
        (EQUAL X "MINUS"))
    (CONS (CONS "ONE-OF" (CONS "NUMBERP" "NIL"))
          "NIL")
(IF (EQUAL X "CONS")
    (CONS (CONS "NONE-OF" "NIL")
          (CONS (CONS "NONE-OF" "NIL")
                "NIL"))
(IF (EQUAL X "PACK")
    (CONS (CONS "NONE-OF" "NIL")
          "NIL")
    F))).
```

3.3. The Formal Definition of LEXEMES

In this subsection we define a function that takes as its argument a list of numbers and returns a list of "lexemes." Each lexeme is either a positive or negative integer or is a literal atom obtained by PACKing the sequence of character codes denoting the lexeme.

We start by naming and grouping certain ASCII character codes.

```
Definitions.
(ASCII-OPEN-PAREN) = 40              ;code for (
(ASCII-CLOSE-PAREN) = 41             ;code for )
(ASCII-SINGLE-GRITCH) = 39           ;code for '
(ASCII-SPACE) = 32                   ;code for <space>
(ASCII-NEWLINE) = 141                ;Lisp Machine code for <newline>
(ASCII-CARRIAGE-RETURN) = 13         ;code for <cr>
(ASCII-LINEFEED) = 10                ;code for <lf>
(ASCII-PLUS-SIGN) = 43               ;code for +
(ASCII-MINUS-SIGN) = 45              ;code for -
(ASCII-DOT) = 46                     ;code for .
(ASCII-SEMI-COLON) = 59              ;code for ;
Definition.
(PARENP N) = (OR (EQUAL N (ASCII-OPEN-PAREN))
                 (EQUAL N (ASCII-CLOSE-PAREN)))
Definition.
(WHITEP N) = (OR (EQUAL N (ASCII-SPACE))
              (OR (EQUAL N (ASCII-NEWLINE))
               (OR (EQUAL N (ASCII-CARRIAGE-RETURN))
                   (EQUAL N (ASCII-LINEFEED)))))
Definition.
(ALPHABETICP N) = (AND (LESSP 64 N) (LESSP N 91))
Definition.
(DIGITP N) = (AND (LESSP 47 N) (LESSP N 58))
Definition.
(SIGNP N) = (OR (EQUAL N (ASCII-PLUS-SIGN))
                (EQUAL N (ASCII-MINUS-SIGN)))
```

The lexical analyzer uses white space, parentheses, certain occurrences of the single quote mark, and semicolon to break the input stream into lexemes. The analyzer accumulates into a list the character codes of each lexeme, in reverse order. Those lists having the syntax of an optionally signed nonempty sequence of digits optionally followed by a decimal point are parsed into positive or negative integers. The function NUMERALP recognizes such lists, using NUMERALP1 to recognize optionally signed nonempty sequences of digits.

```
Definition.
(NUMERALP1 A)
  =
(IF (NLISTP A)
    F
    (AND (DIGITP (CAR A))
         (OR (NLISTP (CDR A))
             (OR (AND (SIGNP (CAR (CDR A)))
                      (NLISTP (CDR (CDR A))))
                 (NUMERALP1 (CDR A))))))

Definition.
(NUMERALP A) = (AND (LISTP A)
                    (IF (EQUAL (CAR A) (ASCII-DOT))
                        (NUMERALP1 (CDR A))
                        (NUMERALP1 A)))
```

(GEN-INTEGER A 1 0) returns the positive or negative integer denoted by A, provided A is a NUMERALP.

```
Definition.
(GEN-INTEGER A SHIFT N)
  =
(IF (NLISTP A)
    N
    (IF (EQUAL (CAR A) (ASCII-DOT))
        (GEN-INTEGER (CDR A) SHIFT N)
        (IF (EQUAL (CAR A) (ASCII-PLUS-SIGN))
            N
            (IF (EQUAL (CAR A) (ASCII-MINUS-SIGN))
                (MINUS N)
                (GEN-INTEGER (CDR A)
                             (TIMES 10 SHIFT)
                             (PLUS N
                              (TIMES SHIFT
                                     (DIFFERENCE (CAR A) 48))))))))
```

Those lexemes not parsed as numbers are treated as literal atoms obtained by PACKing up the list of characters typed (using 0 as the final CDR). Since the characters are accumulated in reverse order, they must be reversed before being PACKed.

```
Definition.
(REVPNAME A PNAME)
  =
(IF (NLISTP A)
    PNAME
    (REVPNAME (CDR A) (CONS (CAR A) PNAME)))
```

GEN-LEXEME generates each lexeme, given the list of character codes accumulated.

```
Definition.
(GEN-LEXEME A) = (IF (NUMERALP A)
                     (GEN-INTEGER A 1 0)
                     (PACK (REVPNAME A 0)))
```

Certain lexemes cannot be written down using our quotation mark convention because they are not the quotations of variable or function symbols. We therefore define functions to permit us to refer to these lexemes more conveniently.

```
Definitions.
(OPEN-PAREN) = (PACK (CONS (ASCII-OPEN-PAREN) 0))
(CLOSE-PAREN) = (PACK (CONS (ASCII-CLOSE-PAREN) 0))
(SINGLE-GRITCH) = (PACK (CONS (ASCII-SINGLE-GRITCH) 0))
(DOT) = (PACK (CONS (ASCII-DOT) 0))
```

EMIT is used to add a new lexeme to the emerging stream of lexemes. The first argument is the accumulated list of character codes and the second is the rest of the lexemes. If the first argument is 0 it means no character codes were accumulated since the last lexeme was emitted.

```
Definition.
(EMIT PNAME LST) = (IF (EQUAL PNAME 0)
                       LST
                       (CONS (GEN-LEXEME PNAME) LST))
```

IGNORE-COMMENT scans the input stream until it has passed a newline or carriage return/linefeed.

```
Definition.
(IGNORE-COMMENT STREAM)
   =
(IF (NLISTP STREAM)
    STREAM
    (IF (EQUAL (CAR STREAM) (ASCII-NEWLINE))
        (CDR STREAM)
        (IF (AND (EQUAL (CAR STREAM)
                        (ASCII-CARRIAGE-RETURN))
                 (AND (LISTP (CDR STREAM))
                      (EQUAL (CAR (CDR STREAM))
                             (ASCII-LINEFEED))))
            (CDR (CDR STREAM))
            (IGNORE-COMMENT (CDR STREAM)))))
```

LEXEMES is the lexical analyzer. The first argument is the list of input character codes. The second argument is the list of character codes accumulated for the current lexeme thus far. (LEXEMES STREAM 0) is the list of lexemes.

```
Definition.
(LEXEMES STREAM PNAME)
   =
(IF (NLISTP STREAM)
    (EMIT PNAME "NIL")
(IF (EQUAL (CAR STREAM) (ASCII-SEMI-COLON))
    (EMIT PNAME
          (LEXEMES (IGNORE-COMMENT (CDR STREAM))
                   0))
(IF (AND (EQUAL (CAR STREAM) (ASCII-SINGLE-GRITCH))
         (EQUAL PNAME 0))
    (EMIT (CONS (CAR STREAM) 0)
          (LEXEMES (CDR STREAM) 0))
(IF (PARENP (CAR STREAM))
    (EMIT PNAME
          (EMIT (CONS (CAR STREAM) 0)
                (LEXEMES (CDR STREAM) 0)))
(IF (WHITEP (CAR STREAM))
    (EMIT PNAME (LEXEMES (CDR STREAM) 0))
    (LEXEMES (CDR STREAM)
             (CONS (CAR STREAM) PNAME)))))))
```

We illustrate LEXEMES by exhibiting a few theorems about it.

```
(LEXEMES |(ABC DEF)| 0) = (CONS (OPEN-PAREN)
                           (CONS "ABC"
                            (CONS "DEF"
                             (CONS (CLOSE-PAREN) "NIL"))))

(LEXEMES |X(A-B)Z| 0) = (CONS "X"
                         (CONS (OPEN-PAREN)
                          (CONS "A-B"
                           (CONS (CLOSE-PAREN)
                            (CONS "Z" "NIL")))))

(LEXEMES |'A ''B C'D| 0) = (CONS (SINGLE-GRITCH)
                            (CONS "A"
                             (CONS (SINGLE-GRITCH)
                              (CONS (SINGLE-GRITCH)
                               (CONS "B"
                                (CONS
                                 (PACK (CONS 67 (CONS 39 (CONS 68 0))))
                                 "NIL"))))))
```

```
    (LEXEMES |A;COMMENT
B|                  0)      = (CONS "A" (CONS "B" "NIL"))

    (LEXEMES |A.B . C| 0)   = (CONS (PACK (CONS 65 (CONS 46 (CONS 66 0))))
                                (CONS (DOT)
                                 (CONS "C" "NIL")))

    (LEXEMES |-12 3. 4-5 6.7| 0)
      =
    (CONS -12.
     (CONS 3.
      (CONS (PACK (CONS 52 (CONS 45 (CONS 53 0))))
       (CONS (PACK (CONS 54 (CONS 46 (CONS 55 0))))
             "NIL"))))
```

3.4. The Formal Definition of PARSE and READ

We now define the function that attempts to parse a list of lexemes into an "s-expression." We say x is an s-expression if and only if either x is a NUMBERP, a NEGATIVEP, a LITATOM whose UNPACK is a CONS nest around 0 of a sequence of ASCII codes, or a LISTP whose CAR and CDR are both recursively s-expressions.

Our parser takes two arguments. The first is a list of lexemes. The second is a list used as a pushdown stack on which lists are accumulated. Each element of the stack is called a "frame" and is itself a list of three items. Whenever the parser encounters an open parenthesis a new frame is pushed on the stack and parsing continues with the character after the open parenthesis. One of the items in the frame collects the s-expressions that are the elements of the list. When the s-expression is completely assembled that stack frame is popped and the s-expression is added to the end of the list being assembled in the newly exposed frame. When a single gritch is read, a count in the frame, initially 0 for each element, is incremented by 1. When the next element of the list is added it is first embedded in as many QUOTE expressions as single gritches preceded it. When the dot lexeme is read, a flag in the frame is set and the next time an s-expression is added to the list being assembled it is put into the final CDR instead of added as the last element.

Here is the function that adds a new frame to the stack.

```
Definition.
(PUSH-FRAME STACK)
  =
(CONS (CONS "NIL" (CONS 0 (CONS F "NIL")))
      STACK)
```

The following three functions return the three items in the top-most frame of the stack.

```
Definition.
(LIST-BEING-ASSEMBLED STACK) = (CAR (CAR STACK))

Definition.
(QUOTE-CNT STACK) = (CAR (CDR (CAR STACK)))

Definition.
(DOT-FLG STACK) = (CAR (CDR (CDR (CAR STACK))))
```

The following function increments the count of single gritches read.

```
Definition.
(BUMP-QUOTE-CNT STACK)
  =
(CONS (CONS (LIST-BEING-ASSEMBLED STACK)
            (CONS (ADD1 (QUOTE-CNT STACK))
                  (CONS (DOT-FLG STACK) "NIL")))
      (CDR STACK))
```

The next two functions turn on and off the flag signalling that a dot has been read.

```
Definition.
(SET-DOT-FLG STACK)
  =
(CONS (CONS (LIST-BEING-ASSEMBLED STACK)
            (CONS (QUOTE-CNT STACK)
                  (CONS T "NIL")))
      (CDR STACK))
```

```
Definition.
(UNSET-DOT-FLG STACK)
  =
(CONS (CONS (LIST-BEING-ASSEMBLED STACK)
            (CONS (QUOTE-CNT STACK)
                  (CONS F "NIL")))
      (CDR STACK))
```

KWOTEN is the function used to embed each s-expression in QUOTEs.

```
Definition.
(KWOTEN N X)
  =
(IF (ZEROP N)
    X
    (CONS "QUOTE"
          (CONS (KWOTEN (SUB1 N) X) "NIL")))
```

The next function adds its first argument to the list being assembled in the top frame of the stack, taking account of the number of gritches that preceded it and whether the dot flag is set. Note that the function resets the quote count to 0 in anticipation of the processing of the next element of the list.

```
Definition.
(ADD-ELEMENT X STACK)
  =
(CONS (CONS (IF (DOT-FLG STACK)
                (APPEND (LIST-BEING-ASSEMBLED STACK)
                        (KWOTEN (QUOTE-CNT STACK) X))
                (APPEND (LIST-BEING-ASSEMBLED STACK)
                        (CONS (KWOTEN (QUOTE-CNT STACK) X)
                              "NIL")))
            (CONS 0
                  (CONS (DOT-FLG STACK) "NIL")))
      (CDR STACK))
```

Here, finally, is the parser. The top-level call of the parser should have a stack with one empty frame on it. The deepest stack frame is treated specially by PARSE: as soon as an element has been added to it, parsing stops and the element is returned.

If the parser encounters ill-formed syntax -- e.g., unmatched parentheses, illegal uses of the dot notation, or unnecessary terminal lexemes after the completion of the parsing -- it returns F.

```
Definition.
(PARSE L STACK)
   =
(IF (NLISTP L)
    F
(IF (EQUAL (CAR L) (OPEN-PAREN))
    (PARSE (CDR L) (PUSH-FRAME STACK))
(IF (EQUAL (CAR L) (CLOSE-PAREN))
    (IF (OR (NLISTP STACK)
            (NLISTP (CDR STACK)))
        F
        (IF (AND (DOT-FLG (CDR STACK))
                 (OR (NLISTP (CDR L))
                     (NOT (EQUAL (CAR (CDR L)) (CLOSE-PAREN)))))
            F
            (IF (NLISTP (CDR (CDR STACK)))
                (IF (LISTP (IF (DOT-FLG (CDR STACK))
                               (CDR (CDR L))
                               (CDR L)))
                    F
                    (CAR (LIST-BEING-ASSEMBLED
                           (ADD-ELEMENT
                             (LIST-BEING-ASSEMBLED STACK)
                             (CDR STACK)))))
                (PARSE (CDR L)
                       (ADD-ELEMENT (LIST-BEING-ASSEMBLED STACK)
                                    (CDR STACK))))))
(IF (EQUAL (CAR L) (SINGLE-GRITCH))
    (PARSE (CDR L) (BUMP-QUOTE-CNT STACK))
(IF (EQUAL (CAR L) (DOT))
    (IF (OR (NLISTP STACK)
            (NLISTP (LIST-BEING-ASSEMBLED STACK)))
        F
        (IF (DOT-FLG STACK)
            F
            (IF (NOT (ZEROP (QUOTE-CNT STACK)))
                F
                (PARSE (CDR L) (SET-DOT-FLG STACK)))))
    (IF (NLISTP STACK)
        F
        (IF (AND (DOT-FLG STACK)
                 (OR (NLISTP (CDR L))
                     (NOT (EQUAL (CAR (CDR L)) (CLOSE-PAREN)))))
            F
            (IF (NLISTP (CDR STACK))
                (IF (LISTP (CDR L))
                    F
                    (CAR (LIST-BEING-ASSEMBLED (ADD-ELEMENT (CAR L) STACK))))
                (PARSE (CDR L)
                       (UNSET-DOT-FLG (ADD-ELEMENT (CAR L) STACK))))))))))
```

The reader is the composition of the parser and the lexical analyzer.

```
Definition.
(READ STREAM) = (PARSE (LEXEMES STREAM 0)
                       (PUSH-FRAME "NIL"))
```

We now illustrate READ by exhibiting some theorems about it:

```
(READ |(A (B C) D)|) = (CONS "A"
                             (CONS (CONS "B" (CONS "C" "NIL"))
                                   (CONS "D" "NIL")))
```

```
(READ |(A 'B C)|)     = (CONS "A"
                              (CONS (CONS "QUOTE" (CONS "B" "NIL"))
                                    (CONS "C" "NIL")))

(READ |(A B . C)|)    = (CONS "A" (CONS "B" "C"))

(READ |(A . (B . (C . ())))|)
                      = (CONS "A"
                              (CONS "B" (CONS "C" "NIL")))

(READ |'((A . 1)(B . 2))|)
                      = (CONS "QUOTE"
                          (CONS (CONS (CONS "A" 1)
                           (CONS (CONS "B" 2) "NIL"))
                                "NIL"))

(READ |(PLUS I (TIMES 33 J) (LOOKUP 'X ALIST))|)
                      =
(CONS "PLUS"
      (CONS "I"
            (CONS (CONS "TIMES"
                        (CONS 33 (CONS "J" "NIL")))
                  (CONS (CONS "LOOKUP"
                              (CONS (CONS "QUOTE"
                                          (CONS "X" "NIL"))
                                    (CONS "ALIST" "NIL")))
                        "NIL")))))
```

3.5. The Formal Definition of TRANSLATE and ISYNTAX

ISYNTAX is the composition of a function called TRANSLATE and the function READ above. Almost all of this section is devoted to the definition of TRANSLATE and its subfunctions. TRANSLATE takes as input an s-expression and produces either F or the primitive quotation of a formal term.

Roughly speaking, TRANSLATE transforms LITATOMs into themselves, provided they have the syntax of our variable symbols, and transforms s-expressions of the form (CONS fn (CONS arg_1 ... (CONS arg_n "NIL"))) into (CONS fn (CONS arg'_1 ... (CONS arg'_n "NIL"))), where arg'_i is the translation of arg_i, provided fn is the quotation of a function symbol of arity n. However, there are many special cases in which more elaborate transformations are performed. The most complicated involve the extended QUOTE notation for denoting explicit values and the handling of FOR expressions.

We first define the function SYMBOLP which recognizes when a LITATOM has the syntax of the symbols in our logic, i.e., is a sequence of alphanumeric characters or hyphens, beginning with an alphabetic character.

```
Definition.
(LEGAL-CHAR-CODE-SEQ1 L)
  =
(IF (NLISTP L)
    T
    (AND (OR (ALPHABETICP (CAR L))
             (OR (DIGITP (CAR L))
                 (EQUAL (CAR L) (ASCII-MINUS-SIGN))))
         (LEGAL-CHAR-CODE-SEQ1 (CDR L))))
```

```
Definition.
(LEGAL-CHAR-CODE-SEQ L)
  =
(AND (LISTP L)
     (AND (EQUAL (CDR (LAST L)) 0)
          (AND (ALPHABETICP (CAR L))
               (LEGAL-CHAR-CODE-SEQ1 (CDR L)))))
```

```
Definition.
(SYMBOLP X) = (AND (LITATOM X)
                   (LEGAL-CHAR-CODE-SEQ (UNPACK X)))
```

TRANSLATE processes the submitted s-expression top-down, checking that each subexpression is legal in the context in which it occurs. As it processes each legal subexpression it CONSes together the primitive quotation of the formal term represented. However, if it encounters an illegal subexpression it must return F as the top-level answer. Thus, instead of using CONS to construct the quotation, TRANSLATE uses FCONS below.

```
Definition.
(FCONS X Y) = (IF (AND X Y) (CONS X Y) F)
```

Perhaps the most complicated part of TRANSLATE is the transformation of QUOTEd expressions. TRANSLATE transforms an input of the form (CONS "QUOTE" (CONS evg "NIL")) into the quotation of an explicit value, provided evg ("explicit value guts") has certain properties.

For example, if evg is an integer, the QUOTE-expression is translated into the primitive quotation of a nest of ADD1's around (ZERO), possibly with a top-level MINUS.

If evg is a LITATOM satisfying the restrictions on symbols, the QUOTE-expression denotes a PACK expression. For example, the result of READing |(QUOTE ABC)| is TRANSLATEd into the primitive quotation of the PACK expression we abbreviate as "ABC" in the formal syntax: |(PACK (CONS 65 (CONS 66 (CONS 67 0))))|.

However, not all LITATOM evgs denote PACK expressions; we use two of the non-symbol LITATOMs to stand for T and F. The two LITATOMs are those produced by READing |*1*TRUE| and |*1*FALSE| and are returned by the functions EVG-TRUE and EVG-FALSE below.

If evg is a LISTP, e.g., the result of READing |(ABC . DEF)|, it represents a CONS, e.g., (CONS "ABC" "DEF"), provided both the CAR and the CDR are evgs.

If evg is a LISTP whose CAR is a certain mark called the EVG-QUOTE-MARK, it represents a nonprimitive shell object or "unusual" primitive ones, such as non-symbol LITATOMs. The mark is the non-symbol LITATOM produced by READing |*1*QUOTE|.

The complicated nature of our representation of explicit values stems from two desires. First, for efficiency in the theorem prover, we have arranged for there to be only one way to represent every explicit value as a QUOTEd evg. Second, we have arranged for the quotation of a term to be produced by embedding the internal representation of the term in a QUOTE expression, permitting the efficient use of "meta" functions. These issues are dealt with at length in [meta].

We now begin defining the functions to manipulate evgs. ADD1-NEST returns the quotation of the formal term denoted by a nonnegative integer.

```
Definition.
(ADD1-NEST N)
   =
(IF (ZEROP N)
    (CONS "ZERO" "NIL")
    (CONS "ADD1"
          (CONS (ADD1-NEST (SUB1 N)) "NIL")))
```

Here are the non-symbol LITATOMs we use in evgs.

```
Definition.
(EVG-TRUE)
   =
(PACK (CONS 42
       (CONS 49
        (CONS 42
              (CONS 84 (CONS 82 (CONS 85 (CONS 69 0))))))))

Definition.
(EVG-FALSE)
   =
(PACK (CONS 42
       (CONS 49
        (CONS 42
              (CONS 70 (CONS 65 (CONS 76 (CONS 83 (CONS 69 0)))))))))

Definition.
(EVG-QUOTE-MARK)
   =
(PACK (CONS 42
       (CONS 49
        (CONS 42
              (CONS 81 (CONS 85 (CONS 79 (CONS 84 (CONS 69 0)))))))))
```

In order for an evg to represent an explicit value it is necessary that its components represent explicit values of the appropriate type. The following functions are used to check type agreement.

```
Definition.
(SHELL-TYPE FN) = (IF (SHELL-BTM-TYPE FN)
                      (SHELL-BTM-TYPE FN)
                      (IF (SHELL-CONS-TYPE FN)
                          (SHELL-CONS-TYPE FN)
                          F))

Definition.
(SHELL-TYPE-OKP FN RESTRICTION)
   =
(IF (EQUAL (CAR RESTRICTION) "ONE-OF")
    (MEMBER (SHELL-TYPE FN)
            (CDR RESTRICTION))
    (NOT (MEMBER (SHELL-TYPE FN)
                 (CDR RESTRICTION))))
```

SHELL-TYPES-OKP takes as its first argument the quotations of n explicit value terms and as its second argument a list of n type restrictions. The function checks that each explicit value term satisfies the corresponding type restriction.

```
Definition.
(SHELL-TYPES-OKP TERMS RESTRICTIONS)
   =
(IF (NLISTP TERMS)
    T
    (AND (LISTP (CAR TERMS))
         (AND (SHELL-TYPE-OKP (CAR (CAR TERMS))
                              (CAR RESTRICTIONS))
              (SHELL-TYPES-OKP (CDR TERMS)
                               (CDR RESTRICTIONS)))))
```

Here is the function that transforms X into the primitive quotation of an explicit value, or else returns F signifying that X is not an evg. If FLG is "LIST" X is considered as a list of evgs instead of as a single evg.

```
Definition.
(EVG FLG X)
   =
(IF (EQUAL FLG "LIST")
    (IF (NLISTP X)
        "NIL"
        (FCONS (EVG T (CAR X))
               (EVG "LIST" (CDR X))))
(IF (NLISTP X)
    (IF (NUMBERP X) (ADD1-NEST X)
    (IF (NEGATIVEP X)
        (CONS "MINUS"
              (CONS (ADD1-NEST (NEGATIVE-GUTS X))
                    "NIL"))
    (IF (EQUAL X (EVG-TRUE)) (CONS "TRUE" "NIL")
    (IF (EQUAL X (EVG-FALSE)) (CONS "FALSE" "NIL")
    (IF (SYMBOLP X)
        (CONS "PACK"
              (CONS (EVG T (UNPACK X))
                    "NIL"))
        F)))))
(IF (EQUAL (CAR X) (EVG-QUOTE-MARK))
    (IF (AND (LISTP (CDR X))
         (AND (EQUAL (CDR (LAST X)) "NIL")
          (AND (EQUAL (LENGTH (CDR (CDR X)))
                      (ARITY (CAR (CDR X))))
           (AND (EVG "LIST" (CDR (CDR X)))
            (AND (OR (SHELL-BTM-TYPE (CAR (CDR X)))
                     (AND (SHELL-CONS-TYPES (CAR (CDR X)))
                          (SHELL-TYPES-OKP (EVG "LIST" (CDR (CDR X)))
                                           (SHELL-CONS-TYPES (CAR (CDR X))))))
                 (IF (EQUAL (CAR (CDR X)) "PACK")
                     (NOT (LEGAL-CHAR-CODE-SEQ (CAR (CDR (CDR X)))))
                     (IF (EQUAL (CAR (CDR X)) "MINUS")
                         (EQUAL (CAR (CDR (CDR X))) 0)
                         (NOT (OR (EQUAL (CAR (CDR X)) "ADD1")
                                  (OR (EQUAL (CAR (CDR X)) "ZERO")
                                      (EQUAL (CAR (CDR X)) "CONS"))))))))))))
        (CONS (CAR (CDR X))
              (EVG "LIST" (CDR (CDR X))))
        F)
    (FCONS "CONS"
           (FCONS (EVG T (CAR X))
                  (FCONS (EVG T (CDR X)) "NIL"))))))
```

This completes the development of the functions for processing evgs.

The next function is the analogue of our notion of the "fn nest around b for s." If FN is the LITATOM corresponding to fn and L is a list of the primitive quotations of the terms $t_1, \ldots, t_n$, then (MAKE-TREE FN L) is the primitive quotation of (fn t_1 ... (fn t_{n-1} t_n)...). If $n<2$, the function returns F.

```
Definition.
(MAKE-TREE FN L)
  =
(IF (NLISTP L)
    F
    (IF (NLISTP (CDR L))
        F
        (IF (NLISTP (CDR (CDR L)))
            (FCONS FN
                   (FCONS (CAR L)
                          (FCONS (CAR (CDR L)) "NIL")))
            (FCONS FN
                   (FCONS (CAR L)
                          (FCONS (MAKE-TREE FN (CDR L))
                                 "NIL"))))))
```

Our implemented notation includes the LISP convention for abbreviating nests of CARs and CDRs with such function symbols as CADR, CADDR, etc. The following functions are used to implement this feature.

CAR-CDRP recognizes those literal atoms which are written down with C as the first character, R as the last, and A's and D's in between. The ASCII codes for A, C, D, and R are 65, 67, 68, and 82.

```
Definition.
(CAR-CDRP1 L)
  =
(IF (NLISTP L)
    F
    (IF (NLISTP (CDR L))
        (AND (EQUAL (CAR L) 82)
             (EQUAL (CDR L) 0))
        (AND (OR (EQUAL (CAR L) 65)
                 (EQUAL (CAR L) 68))
             (CAR-CDRP1 (CDR L)))))
```

```
Definition.
(CAR-CDRP X) = (AND (LITATOM X)
                    (AND (LISTP (UNPACK X))
                         (AND (EQUAL (CAR (UNPACK X)) 67)
                              (CAR-CDRP1 (CDR (UNPACK X))))))
```

This function constructs the quotation of the term denoted by a term beginning with a CAR-CDRP symbol.

```
Definition.
(CAR-CDR-NEST L X)
  =
(IF (OR (NLISTP L) (NLISTP (CDR L)))
    X
    (IF (EQUAL (CAR L) 65)
        (CONS "CAR"
              (CONS (CAR-CDR-NEST (CDR L) X) "NIL"))
        (CONS "CDR"
              (CONS (CAR-CDR-NEST (CDR L) X) "NIL"))))
```

We now move on to the transformation of FOR expressions. We first define convenient "accessors" for the components of the FOR term. Recall that we permit 5, 6, and 7 argument versions of FOR.

```
Definition.
(ABBREVIATED-FOR-VAR X) = (CAR (CDR X))

Definition.
(ABBREVIATED-FOR-RANGE X) = (CAR (CDR (CDR (CDR X))))

Definition.
(ABBREVIATED-FOR-WHEN X)
   =
(IF (EQUAL (LENGTH X) 8)
    (CAR (CDR (CDR (CDR (CDR (CDR X))))))
    "T")

Definition.
(ABBREVIATED-FOR-OP X)
   =
(IF (EQUAL (LENGTH X) 8)
    (CAR (CDR (CDR (CDR (CDR (CDR (CDR X)))))))
    (CAR (CDR (CDR (CDR (CDR X))))))

Definition.
(ABBREVIATED-FOR-BODY X) = (CAR (LAST X))
```

The next function recognizes those LITATOMs that name the operations handled by FOR.

```
Definition.
(FOR-OPERATIONP X)
   =
(OR (EQUAL X "ADD-TO-SET")
    (OR (EQUAL X "ALWAYS")
        (OR (EQUAL X "APPEND")
            (OR (EQUAL X "COLLECT")
                (OR (EQUAL X "COUNT")
                    (OR (EQUAL X "DO-RETURN")
                        (OR (EQUAL X "EXISTS")
                            (OR (EQUAL X "MAX")
                                (OR (EQUAL X "MULTIPLY")
                                    (OR (EQUAL X "SUM")
                                        (EQUAL X "UNION")))))))))))
```

We now define the function that recognizes those FORs requiring fancy translation.

```
Definition.
(ABBREVIATED-FORP X)
   =
(AND (LISTP X)
 (AND (EQUAL (CAR X) "FOR")
  (AND (OR (EQUAL (LENGTH X) 8)
           (EQUAL (LENGTH X) 6))
   (AND (SYMBOLP (ABBREVIATED-FOR-VAR X))
    (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) "NIL"))
     (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) "T"))
      (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) "F"))
       (AND (EQUAL (CAR (CDR (CDR X))) "IN")
        (AND (OR (EQUAL (LENGTH X) 6)
                 (EQUAL (CAR (CDR (CDR (CDR (CDR X))))) "WHEN"))
             (FOR-OPERATIONP (ABBREVIATED-FOR-OP X))))))))))
```

One of the things we do with abbreviated FORs is to compute the association list that binds the "free" variables in the conditional expression and body. We keep those alists in alphabetic order. ALPHABETIC-LESSP compares two LITATOMs and determines whether its first argument is alphabetically smaller than its second.

```
Definition.
(ALPHABETIC-LESSP1 L1 L2)
   =
(IF (NLISTP L1)
    T
    (IF (NLISTP L2)
        F
        (IF (LESSP (CAR L1) (CAR L2))
            T
            (IF (EQUAL (CAR L1) (CAR L2))
                (ALPHABETIC-LESSP1 (CDR L1) (CDR L2))
                F))))
```

```
Definition.
(ALPHABETIC-LESSP X Y) = (ALPHABETIC-LESSP1 (UNPACK X)
                                            (UNPACK Y))
```

Below we define an insertion sort function that sorts lists of LITATOMs alphabetically.

```
Definition.
(ALPHABETIC-INSERT X L)
   =
(IF (NLISTP L)
    (CONS X "NIL")
    (IF (ALPHABETIC-LESSP X (CAR L))
        (CONS X L)
        (CONS (CAR L)
              (ALPHABETIC-INSERT X (CDR L)))))
```

```
Definition.
(ALPHABETIZE L)
   =
(IF (NLISTP L)
    L
    (ALPHABETIC-INSERT (CAR L)
                       (ALPHABETIZE (CDR L))))
```

We next define the function that explores the quotation of a term X and collects the set of variable symbols used in it. If FLG is "LIST" X is considered as a list of quotations instead of a single quotation.

```
Definition.
(ALL-VARS FLG X)
   =
(IF (EQUAL FLG "LIST")
    (IF (NLISTP X)
        "NIL"
        (UNION (ALL-VARS T (CAR X))
               (ALL-VARS "LIST" (CDR X))))
    (IF (NLISTP X)
        (CONS X "NIL")
        (ALL-VARS "LIST" (CDR X))))
```

The function MAKE-ALIST1 takes a list of LITATOMs and returns the quotation of the alist in which the quotation of each symbol is bound to the symbol. That is, if VARS is the quotation list of, say, A, B, and C, then (MAKE-ALIST1 VARS) is [(CONS (CONS "A" A) (CONS (CONS "B" B) (CONS (CONS "C" C) "NIL")))]

```
Definition.
(MAKE-ALIST1 VARS)
   =
(IF (NLISTP VARS)
    (EVG T "NIL")
    (CONS "CONS"
          (CONS (CONS "CONS"
                      (CONS (EVG T (CAR VARS))
                            (CONS (CAR VARS) "NIL")))
                (CONS (MAKE-ALIST1 (CDR VARS))
                      "NIL"))))
```

DELETE deletes the first occurrence of its first argument from its second argument. It is used to remove the "indicial" variable of a FOR statement from the list of variables that occur in the conditional and body expressions.

```
Definition.
(DELETE X L)
   =
(IF (NLISTP L)
    L
    (IF (EQUAL X (CAR L))
        (CDR L)
        (CONS (CAR L) (DELETE X (CDR L)))))
```

Here is the function that constructs the alist for abbreviated FORs, given the indicial variable symbol, the conditional expression, and the body.

```
Definition.
(MAKE-ALIST VAR WHEN BODY)
   =
(MAKE-ALIST1 (ALPHABETIZE (DELETE VAR
                                  (UNION (ALL-VARS T WHEN)
                                         (ALL-VARS T BODY)))))
```

We finally define TRANSLATE. If FLG is "LIST", X is considered to be a list of s-expressions to be translated.

```
Definition.
(TRANSLATE FLG X)
   =
(IF (EQUAL FLG "LIST")
    (IF (NLISTP X)
        "NIL"
        (FCONS (TRANSLATE T (CAR X))
               (TRANSLATE "LIST" (CDR X))))
(IF (NLISTP X)
    (IF (NUMBERP X) (EVG T X)
    (IF (NEGATIVEP X) (EVG T X)
    (IF (LITATOM X)
        (IF (EQUAL X "T") (CONS "TRUE" "NIL")
        (IF (EQUAL X "F") (CONS "FALSE" "NIL")
        (IF (EQUAL X "NIL") (EVG T "NIL")
        (IF (LEGAL-CHAR-CODE-SEQ (UNPACK X)) X
            F))))
      F)))
(IF (NOT (EQUAL (CDR (LAST X)) "NIL"))
    F
(IF (EQUAL (CAR X) "QUOTE")
    (IF (AND (LISTP (CDR X))
             (EQUAL (CDR (CDR X)) "NIL"))
        (EVG T (CAR (CDR X)))
        F)
(IF (OR (EQUAL (CAR X) "NIL")
        (OR (EQUAL (CAR X) "T")
            (EQUAL (CAR X) "F")))
    F
(IF (EQUAL (CAR X) "LIST")
    (IF (TRANSLATE "LIST" (CDR X))
        (IF (NLISTP (CDR X))
            (EVG T "NIL")
            (MAKE-TREE "CONS"
                       (APPEND (TRANSLATE "LIST" (CDR X))
                               (CONS (EVG T "NIL")
                                     "NIL"))))
        F)
(IF (CAR-CDRP (CAR X))
    (IF (AND (LISTP (CDR X))
             (AND (NLISTP (CDR (CDR X)))
                  (TRANSLATE T (CAR (CDR X)))))
        (CAR-CDR-NEST (CDR (UNPACK (CAR X)))
                      (TRANSLATE T (CAR (CDR X))))
        F)
(IF (EQUAL (LENGTH (CDR X)) (ARITY (CAR X)))
    (FCONS (CAR X) (TRANSLATE "LIST" (CDR X)))
(IF (EQUAL (CAR X) "FOR")
    (IF (ABBREVIATED-FORP X)
        (FCONS "FOR"
         (FCONS (EVG T (ABBREVIATED-FOR-VAR X))
          (FCONS (TRANSLATE T (ABBREVIATED-FOR-RANGE X))
           (FCONS (EVG T (TRANSLATE T (ABBREVIATED-FOR-WHEN X)))
            (FCONS (EVG T (ABBREVIATED-FOR-OP X))
             (FCONS (EVG T (TRANSLATE T (ABBREVIATED-FOR-BODY X)))
              (FCONS (MAKE-ALIST (ABBREVIATED-FOR-VAR X)
                                 (TRANSLATE T (ABBREVIATED-FOR-WHEN X))
                                 (TRANSLATE T (ABBREVIATED-FOR-BODY X)))
                     "NIL")))))))
        F)
(IF (AND (LESSP 2 (LENGTH (CDR X)))
         (OR (EQUAL (CAR X) "AND")
          (OR (EQUAL (CAR X) "OR")
           (OR (EQUAL (CAR X) "PLUS")
               (EQUAL (CAR X) "TIMES")))))
    (MAKE-TREE (CAR X) (TRANSLATE "LIST" (CDR X)))
    F))))))))))
```

The implemented syntax is defined by the function ISYNTAX:

```
Definition.
(ISYNTAX STREAM) = (TRANSLATE T (READ STREAM))
```

References

1. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.